

Package ‘bvpSolve’

July 1, 2009

Version 1.0

Title Solvers for boundary value problems of ordinary differential equations

Author Karline Soetaert <k.soetaert@nioo.knaw.nl>

Maintainer Karline Soetaert <k.soetaert@nioo.knaw.nl>

Depends R (>= 2.01), rootSolve, deSolve

Description Functions that solve boundary value problems (BVP) of systems of ordinary differential equations (ODE). The functions provide an interface to the FORTRAN function twpbvp and an R-implementation of the shooting method.

License GPL

LazyData yes

Repository CRAN

Repository/R-Forge/Project bvpsolve

Repository/R-Forge/Revision 27

Date/Publication 2009-07-01 14:47:11

R topics documented:

bvpSolve-package	2
bvpshoot	3
bvptwp	8

Index	14
--------------	-----------

bvpSolve-package *Solvers for Boundary Value Problems (BVP) of Ordinary Differential Equations*

Description

Functions that solve boundary value problems of a system of stiff ordinary differential equations (ODE)

The functions provide an interface to the FORTRAN code twpbvp written by J.R. Cash and M.H. Wright.

and also implements a shooting method

Details

Package: bvpSolve
Type: Package
Version: 1.0
Date: 2009-06-12
License: GNU Public License 2 or above

The system of ODE's is written as an R function, similar as the initial value problems that are solved by integration routines from package deSolve.

Author(s)

Karline Soetaert (Maintainer)

References

J.R. Cash and M.H. Wright, (1991) A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation, SIAM J. Sci. Stat. Comput. 12, 971-989.

See Also

[bvptwp](#), a MIRK method to solve two-point boundary value problems (Cash and Wright, 1991).

[bvpshoot](#), a shooting method, using solvers from packages deSolve and rootSolve.

Examples

```
## Not run:  
## show examples (see respective help pages for details)  
example(bvptwp)  
example(bvpshoot)  
  
## open the directory with examples
```

```

browseURL(paste(system.file(package = "bvpSolve"), "/examples", sep = ""))

## show package vignette with how to use bvpSolve
## + source code of the vignette
vignette("bvpSolve")
edit(vignette("bvpSolve"))

## show directory with source code of the vignette
browseURL(paste(system.file(package = "bvpSolve"), "/doc", sep = ""))
## End(Not run)

```

 bvpsshoot

Solver for boundary value problems of ordinary differential equations

Description

Solves a boundary value problem of a system of ordinary differential equations using the shooting method.

Usage

```

bvpsshoot(yini, x, func, yend, parms=NULL,
          guess=NULL, extra=NULL, atol=1e-8, rtol=1e-8,
          maxiter=100, positive = FALSE, method="lsoda", ...)

```

Arguments

yini	<p>either a vector with the initial (state) values for the ODE system, or a function that calculates the initial condition.</p> <p>If yini has a name attribute, the names will be used to label the output matrix. If yini is a function, it will be called as: <code>yini(y, parms, ...)</code>;</p> <p>if yini is a vector then use NA for an initial value which is not available.</p>
x	<p>sequence of the independent variable for which output is wanted; the first value of x must be the initial value (at which yini is defined), the final value the end condition (at which yend is defined).</p>
func	<p>an R-function that computes the values of the derivatives in the ODE system (the model definition) at x. func must be defined as: <code>yprime = func(x, y, parms, ...)</code>. x is the current point of the independent variable in the integration, y is the current estimate of the (state) variables in the ODE system. If the initial values yini has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.</p> <p>The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to x, and whose next elements are global values that are required at each point in x.</p>
yend	<p>either a vector with the final (state) values for the ODE system, or NULL; if yend is a vector use NA for a final value which is not available.</p>

<code>parms</code>	parameters passed to <code>func</code> .
<code>guess</code>	guess for the value(s) of the unknown initial conditions; i.e. one value for each NA in <code>yini</code> . The length of <code>guess</code> should thus equal the number of NAs in <code>yini</code> . If not provided, a value = 0 is assumed for each NA and a warning printed.
<code>extra</code>	if too many boundary conditions are given, then an extra parameter can be estimated. <code>extra</code> should contain the initial guess.
<code>atol</code>	absolute error tolerance, either a scalar or a vector, one value for each unknown element - passed to function <code>multiroot</code> - see help of this function.
<code>rtol</code>	relative error tolerance, either a scalar or a vector, one value for each unknown element - passed to function <code>multiroot</code> - see help of this function.
<code>maxiter</code>	the maximal number of iterations allowed in the root solver.
<code>positive</code>	set to TRUE if state variables have to be positive numbers.
<code>method</code>	the integration method used, one of ("lsoda", "lsode", "lsodes", "vode", "euler", "rk4", "ode23" or "ode45").
<code>...</code>	additional arguments passed to the integrator (and possibly the model functions).

Details

This is a simple implementation of the shooting method to solve boundary value problems of ordinary differential equations.

A boundary value problem does not have all initial values of the state variable specified. Rather some conditions are specified at the end of the integration interval.

The shooting method, is a root-solving method, where the unknown initial conditions are the unknown values to be solved for; the function value whose root has to be found are the deviations from the final conditions.

Thus, starting with an initial guess of the initial conditions (as provided in `guess`, the ODE model is solved (as an initial value problem), and after termination, the discrepancy of the modeled final conditions with the known final condition is assessed (the cost function). The root of this cost function is to be found.

In `bvpsshoot` one of the integrators from package `deSolve` (as specified with `method`) are used to solve the resulting initial value problem.

Function `multiroot` from package `rootSolve` is used to retrieve the root.

For this method to work, the model should be even determined, i.e. the total number of specified boundary conditions (on both the start and end of the integration interval) should equal the number of boundary value problem equations. An exception is when the number of boundary conditions specified exceeds the number of equations. In this case, extra parameters have to be solved for to make the model even determined.

See examples

Value

A matrix with up to as many rows as elements in `times` and as many columns as elements in `yini` plus the number of "global" values returned in the second element of the return from `func`, plus an additional column (the first) for the `x`-value.

There will be one row for each element in x unless the solver returns with an unrecoverable error.

If y has a names attribute, it will be used to label the columns of the output value.

The output will have the attribute `roots`, which returns the value(s) of the root(s) solved for (`root`), the function value (`f.root`), and the number of iterations (`iter`) required to find the root.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

See Also

[bvptwp](#) for the MIRK method
[lsoda](#), [lsode](#), [lsodes](#), [vode](#),
[rk](#), [rkMethod](#) for details about the integration method

Examples

```
#####
# Example 1: simple standard problem
# solve the BVP ODE:
# d2y/dt^2=-3py/(p+t^2)^2
# y(t= -0.1)=-0.1/sqrt(p+0.01)
# y(t=  0.1)= 0.1/sqrt(p+0.01)
# where p = 1e-5
#
# analytical solution y(t) = t/sqrt(p + t^2).
#
# The problem is rewritten as a system of 2 ODEs:
# dy=y2
# dy2=-3p*y/(p+t^2)^2
#####

#-----
# Derivative function
#-----
fun <- function(t,y,pars)
{ dy1 <- -y[2]
  dy2 <- - 3*p*y[1]/(p+t*t)^2
  return(list(c(dy1,
                dy2))) }

# parameter value
p <- -1e-5

# initial and final condition; second conditions unknown
init <- c(-0.1/sqrt(p+0.01), NA)
end <- c(0.1/sqrt(p+0.01), NA)

# Solve bvp
sol <- as.data.frame(bvpshoot(yini=init, x=seq(-0.1,0.1,by=0.001),
```

```

        func=fun, yend=end, guess=1))
plot(sol$time,sol[,2],type="l")

# add analytical solution
curve(x/sqrt(p+x*x),add=TRUE,type="p")

#####
# Example 1b: simple
# solve  $d^2y/dx^2 + 1/x*dy/dx + (1-1/(4x^2))y = \sqrt{x}*\cos(x)$ ,
# on the interval [1,6] and with boundary conditions:
#  $y(1)=1, y(6)=-0.5$ 
#
# Write as set of 2 odes
#  $dy/dx = y^2$ 
#  $dy^2/dx = -1/x*dy/dx - (1-1/(4x^2))y + \sqrt{x}*\cos(x)$ 
#####

f2 <- function(x,y,parms)
{
  dy <- y[2]
  dy2 <- -1/x*y[2]-(1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
  list(c(dy,dy2))
}

x <- seq(1,6,0.1)
sol <- bvpsshoot(yini=c(1,NA), yend=c(-0.5,NA), x=x, func=f2, guess=1)
plot(sol)

# add the analytic solution
curve(0.0588713*cos(x)/sqrt(x)+1/4*sqrt(x)*cos(x)+0.740071*sin(x)/sqrt(x)+
      1/4*x^(3/2)*sin(x),add=TRUE,type="l")

#####
# Example 2 - initial condition is a function of the unknown x
# tubular reactor with axial dispersion
#  $y' = Pe(y' + Ry^n)$   $Pe=1, R=2, n=2$ 
# on the interval [0,1] and with initial conditions:
#  $y'(0)=Pe(y(0)-1), y'(1)=0$ 
#
#  $dy=y^2$ 
#  $dy^2=Pe(dy-Ry^n)$ 
#####

Reactor<-function(x,y,parms)
{
  list(c(y[2],
        Pe*(y[2]+R*(y[1]^n))))
}

Pe <- 1
R <- 2
n <- 2

```

```

yini <- function (x,parms) c(x,Pe*(x-1))
x     <- seq(0,1,by=0.01)
sol<-bvpsshoot(func=Reactor, yend=c(NA,0), y=yini, x=x, extra=1)
plot(sol,main="Reactor")

#####
# Example 3 - final condition is a residual function
# The example for MUSN in Ascher et al., 1995.
# Numerical Solution of Boundary Value Problems for Ordinary Differential
# Equations, SIAM, Philadelphia, PA, 1995.
# MUSN is a multiple shooting code for nonlinear BVPs.
#
# The problem is
#   u' = 0.5*u*(w - u)/v
#   v' = -0.5*(w - u)
#   w' = (0.9 - 1000*(w - y) - 0.5*w*(w - u))/z
#   z' = 0.5*(w - u)
#   y' = -100*(y - w)
# on the interval [0 1] and with boundary conditions:
#   u(0) = v(0) = w(0) = 1, z(0) = -10, w(1) = y(1)
#####

musn <- function(t,Y,pars)
{
  with (as.list(Y),
    {
      du=0.5*u*(w-u)/v
      dv=-0.5*(w-u)
      dw=(0.9-1000*(w-y)-0.5*w*(w-u))/z
      dz=0.5*(w-u)
      dy=-100*(y-w)
      return(list(c(du,dv,dw,dy,dz)))
    })
}

#-----
# Residuals
#-----
res <- function (Y,yini,parms) with (as.list(Y), w-y)

#-----
# Initial values; y= NA= not available
#-----

init <- c(u=1,v=1,w=1,y=NA,z=-10)
sol <-bvpsshoot(y= init, x=seq(0,1,by=0.05), func=musn,
               yend=res, guess=1, atol=1e-10, rtol=0)
pairs(sol, main="MUSN")

#####
# Example 4 - solve also for unknown parameter
# Find the 4th eigenvalue of Mathieu's equation:
# y''+(lam-10cos2t)y=0 on the interval [0,pi]

```

```

# y(0)=1, y'(0)=0 and y'(pi)=0
# The 2nd order problem is rewritten as 2 first-order problems:
# dy=y2
# dy2= -(lam-10cos(2t))*y
#####

mathieu<- function(t,y,lam)
{
  list(c(y[2],-(lam-10*cos(2*t))*y[1]))
}

yini <- c(1,0) # initial condition(y1=1,dy=y2=0)
yend <- c(NA,0) # final condition at pi (y1=NA, dy=0)

# there is one extra parameter to be fitted: "lam"; its initial guess = 15
Sol <- bvpshoot(yini=yini, yend=yend, x=seq(0,pi,by=0.01),
               func=mathieu, guess=NULL, extra=15)
plot(Sol)
attr(Sol,"roots") # root gives the value of "lam" (17.10684)

```

 bvptwp

Solver for two-point boundary value problems of ordinary differential equations, a mono-implicit Runge-Kutta (MIRK) formula

Description

Solves a boundary value problem of a system of ordinary differential equations. This is an implementation of the fortran code twpbvp, based on mono-implicit Runge-Kutta formulae (MIRK) of orders 4, 6 and 8 in a deferred correction framework

written by J.R. Cash and M.H. Wright.

Usage

```

bvptwp(yini=NULL, x, func, yend=NULL, parms=NULL, guess=NULL,
       xguess=NULL, yguess=NULL, jacfunc=NULL, bound=NULL, jacobound=NULL,
       leftbc=NULL, islin=FALSE, nmax=1000, colp= NULL, atol=1e-8, ...)

```

Arguments

yini	either a vector with the initial (state) values for the ODE system, or NULL. If yini is a vector, use NA for an initial value which is not available.
x	sequence of the independent variable for which output is wanted; the first value of x must be the initial value (at which yini is defined), the final value the end condition (at which yend is defined).
func	an R-function that computes the values of the derivatives in the ODE system (the model definition) at x. func must be defined as: <code>yprime = func(x, y, parms, ...)</code> . x is the current point of the independent variable in the integration, y is the current estimate of the (state) variables in the ODE system.

If the initial values `yini` has a `names` attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to `x`, and whose next elements are global values that are required at each point in `x`.

<code>yend</code>	either a vector with the final (state) values for the ODE system, or <code>NULL</code> ; if <code>yend</code> is a vector use <code>NA</code> for a final value which is not available.
<code>parms</code>	parameters passed to <code>func</code> .
<code>guess</code>	guess for the value(s) of the unknown initial conditions; i.e. one value for each <code>NA</code> in <code>yini</code> . The length of <code>guess</code> should thus equal the number of <code>NA</code> s in <code>yini</code> . If not provided, a value = 0 is assumed for each <code>NA</code> and a warning printed.
<code>xguess</code>	Initial grid <code>x</code> , a vector.
<code>yguess</code>	First guess values <code>y</code> , corresponding to initial grid <code>xguess</code> ; a matrix with number of rows=number of equations, and whose number of columns = length of <code>xguess</code> .
<code>jacfunc</code>	<p>jacobian (optional) - an R-function that evaluates the jacobian of <code>func</code> at point <code>x</code>, <code>jacfunc</code> must be defined as: <code>jac = func(x, y, parms, ...)</code>. It should return the partial derivatives of <code>func</code> with respect to <code>y</code>, i.e. $df(i,j) = df_i/dy_j$. See last example.</p> <p>If <code>jacfunc</code> is <code>NULL</code>, then a numerical approximation using differences is used.</p>
<code>bound</code>	<p>boundary function (optional) - only if <code>yini</code> and <code>yend</code> are not available. An R function that evaluates the <code>i</code>-th boundary element at point <code>x</code>. It should be defined as: <code>bound = func(i, y, parms, ...)</code>. It should return the <code>i</code>-th boundary condition. See last example.</p>
<code>jacbound</code>	<p>jacobian of the boundary function (optional) - only if <code>bound</code> is defined. An R function that evaluates the gradient of the <code>i</code>-th boundary element with respect to the state variables, at point <code>x</code>. It should be defined as: <code>jacbound = func(i, x, y, parms, ...)</code>. It should return the gradient of the <code>i</code>-th boundary condition. See last example.</p> <p>If <code>jacbound</code> is <code>NULL</code>, then a numerical approximation using differences is used.</p>
<code>leftbc</code>	only if <code>yini</code> and <code>yend</code> are not available: the number of left boundary conditions.
<code>islin</code>	set to <code>TRUE</code> if the problem is linear - this will speed up the simulation.
<code>nmax</code>	maximal number of subintervals.
<code>colp</code>	number of points per subinterval.
<code>atol</code>	error tolerance, a scalar.
<code>...</code>	additional arguments passed to the model functions.

Details

This is an implementation of the method `twpbvp`, to solve two-point boundary value problems of ordinary differential equations.

A boundary value problem does not have all initial values of the state variable specified. Rather some conditions are specified at the end of the integration interval.

The ODEs and boundary conditions are made available through the user-provided routines, `func` and vectors `yini` and `yend` or (optionally) `bound`.

The corresponding partial derivatives are optionally available through the user-provided routines, `jacfunc` and `jacbound`. Default is that they are automatically generated by `R`, using numerical differences.

The user-requested tolerance is provided through `tol`. The default is $1e^{-6}$.

If the function terminates because the maximum number of subintervals was exceeded, then it is recommended that the program be run again with a larger value for this maximum.

For this method to work, there should be ONLY one solution. This means that the number of specified boundary value problems must equal the number of unspecified initial conditions (so that there is exactly ONE root).

Value

A matrix with up to as many rows as elements in `times` and as many columns as elements in `yini` plus the number of "global" values returned in the second element of the return from `func`, plus an additional column (the first) for the `x`-value.

There will be one row for each element in `x` unless the solver returns with an unrecoverable error.

If `yini` has a `names` attribute, it will be used to label the columns of the output value.

Author(s)

Karline Soetaert <k.soetaert@nioo.knaw.nl>

References

J.R. Cash and M.H. Wright, A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation, *SIAM J. Sci. Stat. Comput.*, 12 (1991) 971-989.

http://www.ma.ic.ac.uk/~jcash/BVP_software

See Also

`bvpshoot` for the shooting method

Examples

```
#####
# Example 1: simple standard problem
# solve the BVP ODE:
# d2y/dt^2=-3py/(p+t^2)^2
```

```

# y(t= -0.1)=-0.1/sqrt(p+0.01)
# y(t= 0.1)= 0.1/sqrt(p+0.01)
# where p = 1e-5
#
# analytical solution y(t) = t/sqrt(p + t^2).
#
# The problem is rewritten as a system of 2 ODEs:
# dy=y2
# dy2=-3p*y/(p+t^2)^2
#####

#-----
# Derivative function
#-----
fun <- function(t,y,pars)
{ dy1 <-y[2]
  dy2 <- - 3*p*y[1]/(p+t*t)^2
  return(list(c(dy1,
                dy2))) }

# parameter value
p <-1e-5

# initial and final condition; second conditions unknown
init <- c(-0.1/sqrt(p+0.01), NA)
end <- c(0.1/sqrt(p+0.01), NA)

# Solve bvp
sol <- as.data.frame(bvptwp(yini=init,x=seq(-0.1,0.1,by=0.001),
                           func=fun, yend=end, guess=1))
plot(sol$time,sol[,2],type="l")

# add analytical solution
curve(x/sqrt(p+x*x),add=TRUE,type="p")

#####
# Example 1b: simple
# solve d2y/dx2 + 1/x*dy/dx + (1-1/(4x^2))y = sqrt(x)*cos(x),
# on the interval [1,6] and with boundary conditions:
# y(1)=1, y(6)=-0.5
#
# Write as set of 2 odes
# dy/dx = y2
# dy2/dx = - 1/x*dy/dx - (1-1/(4x^2))y + sqrt(x)*cos(x)
#####

f2 <- function(x,y,parms)
{
  dy <- y[2]
  dy2 <- -1/x*y[2]-(1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
  list(c(dy,dy2))
}

```

```

x    <- seq(1,6,0.1)
sol  <- bvptwp(yini=c(1,NA),yend=c(-0.5,NA),x=x,func=f2,guess=1)
plot(sol)

# add the analytic solution
curve(0.0588713*cos(x)/sqrt(x)+1/4*sqrt(x)*cos(x)+0.740071*sin(x)/sqrt(x)+
      1/4*x^(3/2)*sin(x),add=TRUE,type="l")

#####
# Problem 2 - solved with specification of boundary, and jacobians
#  $d^4y/dx^4 = R(dy/dx*d^2y/dx^2 - y*dy^3/dx^3)$ 
#  $y(0)=y'(0)=0$ 
#  $y(1)=1, y'(1)=0$ 
#
#  $dy/dx = y^2$ 
#  $dy^2/dx = y^3$  (=d2y/dx2)
#  $dy^3/dx = y^4$  (=d3y/dx3)
#  $dy^4/dx = R*(y^2*y^3 - y*y^4)$ 
#####

f2<- function(x,y,parms,R)
{
  list(c(y[2],y[3],y[4],R*(y[2]*y[3]-y[1]*y[4])) )
}

df2 <- function(x,y,parms,R)
{
  df <- matrix(nr=4,nc=4,byrow=TRUE,data=c(
    0,1,0,0,
    0,0,1,0,
    0,0,0,1,
    -1*R*y[4],R*y[3],R*y[2],-R*y[1]))
}

g2 <- function(i,y,parms,R)
{
  if ( i ==1) return(y[1])
  if ( i ==2) return(y[2])
  if ( i ==3) return(y[1]-1)
  if ( i ==4) return(y[2])
}

dg2 <- function(i,y,parms,R)
{
  if ( i ==1) return(c(1,0,0,0))
  if ( i ==2) return(c(0,1,0,0))
  if ( i ==3) return(c(1,0,0,0))
  if ( i ==4) return(c(0,1,0,0))
}

require(bvpSolve)
init <- c(1,NA)
R    <- 0.01

```

```
sol <- as.data.frame(bvptwp(x=seq(0,1,by=0.01), leftbc=2,  
  func=f2, guess=1,R=R,  
  bound=g2, jacfunc=df2, jacbound=dg2))  
plot(sol[,1:2])
```

Index

*Topic **math**

 bvpshoot, 3

 bvptwp, 8

*Topic **package**

 bvpSolve-package, 2

bvpshoot, 2, 3, 10

bvpSolve (*bvpSolve-package*), 2

bvpSolve-package, 2

bvptwp, 2, 5, 8

lsoda, 5

lsode, 5

lsodes, 5

multiroot, 4

rk, 5

rkMethod, 5

vode, 5